**PostConstruct, PreDestroy and Event Listener**

[**https://medium.com/@gaddamnaveen192/springboot3-lifecycle-replace-postconstruct-and-predestroy-with-modern-solutions-219be6f4e9d5**](https://medium.com/@gaddamnaveen192/springboot3-lifecycle-replace-postconstruct-and-predestroy-with-modern-solutions-219be6f4e9d5)

* @PostConstruct**and**@PreDestroy

In Spring Boot, @PostConstruct and @PreDestroy are lifecycle annotations provided by **Jakarta EE (formerly Java EE)**, specifically from the jakarta.annotation package. These annotations are not part of Spring Boot itself but are supported by Spring's lifecycle management.

* **1. What is**@PostConstruct**?**

@PostConstruct is an annotation in Spring Boot that tells Spring to run a specific method **after the bean is created and dependencies are injected** but **before the bean is used**.

* **When is it used?**

It is used when you want to **initialize some data or perform setup tasks** before your application starts serving requests.

* **Example Without**@PostConstruct**(Problem)**

Imagine you have a UserService that loads users from a database

@Service  
public class UserService {  
 private final UserRepository userRepository;  
 private Map<Long, User> userCache = new HashMap<>();  
  
 public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
 }  
  
 public User getUserById(Long id) {  
 return userCache.get(id); // Returns null if cache is empty  
 }  
}

**Problem**: The userCache is empty when the application starts.

* **Solution With**@PostConstruct

To **preload users into the cache**, we use @PostConstruct

import javax.annotation.PostConstruct;  
import org.springframework.stereotype.Service;  
import java.util.HashMap;  
import java.util.Map;  
  
@Service  
public class UserService {  
 private final UserRepository userRepository;  
 private Map<Long, User> userCache = new HashMap<>();  
  
 public UserService(UserRepository userRepository) {  
 this.userRepository = userRepository;  
 }  
  
 @PostConstruct  
 public void init() {  
 System.out.println("Loading users into cache...");  
 userRepository.findAll().forEach(user -> userCache.put(user.getId(), user));  
 System.out.println("Users loaded successfully!");  
 }  
  
 public User getUserById(Long id) {  
 return userCache.get(id); // Now it works because cache is preloaded  
 }  
}

* **How It Works?**

1. Spring Boot **creates the UserService bean**.
2. It **injects UserRepository into UserService**.
3. The @PostConstruct method init() runs **automatically**, loading users into the cache.
4. Now, getUserById(id) can return users from the cache instead of an empty map

It’s important to note that while @PostConstruct was removed from the javax.annotation package starting with Java 9+, it has been moved to the jakarta.annotation package as part of the Jakarta EE transition. This means that @PostConstruct and @PreDestroy are still available in Spring Boot 3+, just under the jakarta.annotation namespace. Therefore, they can still be used for initialization and cleanup tasks, but now in alignment with the Jakarta EE standard As an alternative, you can use **Spring’s @EventListener** with ApplicationReadyEvent to achieve similar functionality

* **What is**@EventListener(ApplicationReadyEvent.class)**?**
* It is a Spring-specific way to run code **after the application has fully started** and is ready to handle requests.
* Unlike @PostConstruct, which runs after a single bean is initialized, @EventListener(ApplicationReadyEvent.class) runs **once the entire Spring application context is fully initialized**
* **Why Use**@EventListener(ApplicationReadyEvent.class)**?**
* It ensures that your initialization logic runs **only after the entire application is ready**.
* It is useful for tasks like:
* Loading data into the database.
* Starting background processes.
* Validating configurations.
* Sending startup notifications
* **Example: Using**@EventListener(ApplicationReadyEvent.class)

Here’s how you can replace @PostConstruct with @EventListener(ApplicationReadyEvent.class)

import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class MyStartupTask {  
  
 // This method will run when the application is fully started  
 @EventListener(ApplicationReadyEvent.class)  
 public void onApplicationReady() {  
 System.out.println("Application is fully started and ready!");  
 performStartupTasks();  
 }  
  
 private void performStartupTasks() {  
 System.out.println("Performing startup tasks...");  
 // Example: Load data, validate configurations, etc.  
 }  
}

* **Example: Loading Initial Data on Startup**

import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class DataLoader {  
  
 @EventListener(ApplicationReadyEvent.class)  
 public void loadData() {  
 System.out.println("Loading initial data into the database...");  
 // Logic to load data  
 }  
}

* **Example: Sending a Startup Notification**

import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class StartupNotifier {  
  
 @EventListener(ApplicationReadyEvent.class)  
 public void notifyStartup() {  
 System.out.println("Application is ready! Sending notification...");  
 // Logic to send a notification (e.g., email, Slack message)  
 }  
}

* **2. What is**@PreDestroy**in Spring Boot?**

In Spring Boot, @PreDestroy is used to mark a method that should run just **before** a **bean** (an object) is removed from the Spring container, typically when the application shuts down or when the bean is no longer needed.

You can think of it as a **clean-up method** that performs necessary tasks like closing files, releasing resources, or stopping background processes before the application completely shuts down.

* **Real-World Example:**

Imagine you have a **coffee machine** that, after being used, needs to **clean itself**. The cleaning process must happen **before the machine is turned off**, so it doesn’t leave any coffee grounds or water inside. This clean-up happens just before you **switch off** the machine.

In Spring Boot, @PreDestroy is like the cleaning process that ensures things are **properly cleaned up** before shutting down the application.

* **Code Example:**

Let’s say we have a service that opens a file when the application starts, and we want to **close the file** before the application shuts down

import javax.annotation.PreDestroy;  
import org.springframework.stereotype.Service;  
  
@Service  
public class FileService {  
  
 // This simulates opening a file  
 private String fileResource = "File opened";  
  
 // @PreDestroy method to clean up the file resource before the bean is destroyed  
 @PreDestroy  
 public void cleanUp() {  
 System.out.println("Cleaning up the resources before shutdown...");  
 // Simulating closing the file or resource  
 fileResource = null;  
 }  
}

* **What Happens Here:**
* When the application starts, the FileService bean is created, and it opens the file (this is just a simulation).
* When the application is shutting down, the @PreDestroy method cleanUp() is called, which **closes the file** (or in our case, nullifies the resource).
* This ensures that we **don’t leave any resources open** when the application stops

I think you have a doubt, but here @PostConstruct is deprecated.*javax.annotation package starting with Java 9+* Why is @PreDestroy not deprecated?

You’re absolutely right about the removal of javax.annotation.PostConstruct in **Spring Boot 3+**, which is a consequence of the **javax.annotation package being deprecated** in Java 9 . However, **@PreDestroy** is still available for use in Spring Boot 3+ due to some important reasons:

* **Why**@PreDestroy**Is Still Available in Spring Boot 3+:**

1. **Backward Compatibility**: Spring is designed to maintain backward compatibility with existing applications. While @PostConstruct is no longer recommended due to its reliance on javax.annotation (which is removed in Java 9+), **@PreDestroy** continues to work for the time being. It hasn't been deprecated or removed because:

* It doesn’t rely on any special external package for its functionality.
* It is still considered useful for many applications, especially for performing cleanup tasks like closing resources or shutting down threads.

1. **Jakarta EE Transition**: With Java 9+, the **javax.annotation** package was moved to **Jakarta EE**. The @PreDestroy annotation is now part of the **Jakarta Annotations** package, which is the future standard for annotations in Java EE-based applications (like Spring).
2. In Spring 3+, while javax.annotation is removed, Spring **supports Jakarta EE annotations** (such as @PreDestroy) and is transitioning toward them. **Spring 6** (which is aligned with Jakarta EE) is more focused on adopting **Jakarta's annotations** and libraries, which is why @PreDestroy can still be used in Spring Boot 3+ without issue.

* **Alternative to**@PostConstruct**and**@PreDestroy**in Spring Boot 3+:**

If you’re targeting **Spring Boot 3+**, and the @PostConstruct or @PreDestroy annotations are unavailable due to the Jakarta transition, you can use **Spring's own event listener mechanism**.

**For Initialization (**@PostConstruct**alternative):**

Instead of @PostConstruct, you can use Spring's @EventListener with the ApplicationReadyEvent to trigger initialization logic

import org.springframework.boot.context.event.ApplicationReadyEvent;  
import org.springframework.context.event.EventListener;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyService {  
  
 @EventListener(ApplicationReadyEvent.class)  
 public void init() {  
 System.out.println("Application is ready. Performing initialization...");  
 }  
}

* **For Cleanup (**@PreDestroy**alternative):**

If you want to replace @PreDestroy for cleanup logic, you can implement the DisposableBean interface or use the @Bean(destroyMethod = "methodName") approach.

1. **Using DisposableBean**:

import org.springframework.beans.factory.DisposableBean;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyService implements DisposableBean {  
  
 @Override  
 public void destroy() throws Exception {  
 System.out.println("Cleaning up resources before shutdown...");  
 }  
}

2. Using @Bean(destroyMethod = "methodName")

import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class AppConfig {  
  
 @Bean(destroyMethod = "close")  
 public MyService myService() {  
 return new MyService();  
 }  
}  
  
class MyService {  
  
 public void close() {  
 System.out.println("Cleaning up resources...");  
 }  
}

* **Summary:**
* **@PreDestroy** is still available in Spring Boot 3+ because it remains useful for resource cleanup, and the functionality is supported by Jakarta EE annotations.
* **@PostConstruct** is removed in Spring Boot 3+ due to the deprecation and removal of javax.annotation in Java 9+, and it's recommended to use Spring alternatives like @EventListener(ApplicationReadyEvent.class)